Title

Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Polyhedra

Authors

Bennett, Janine
Cook, Richard
Max, Nelson
et al.

Publication Date

2001

Peer reviewed

Preprint
UCRL-JC-143127-REV-1

Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Ployhedra

J. Bennett, R. Cook, N. Max, D. May, and P. Williams

This article was submitted to IEEE Symposium on Parallel and Large-Data Visualization and Graphics, San Diego, CA
October 22 - 23, 2001

U.S. Department of Energy

Lawrence Livermore National Laboratory

July 23, 2001

Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Polyhedra

Janine Bennett[1,2], Richard Cook[1,2], Nelson Max[1,2], Deborah May[2] and Peter Williams[2]

Abstract

This paper discusses our efforts to improve the performance of the high-accuracy (HIAC) volume rendering system, based on cell projection, which is used to display unstructured, scientific data sets for analysis. The parallelization of HIAC, using the pthreads and MPI API's, resulted in significant speedup, but interactive frame rates are not yet attainable for very large data sets.

1 INTRODUCTION

Volume visualization of scientific datasets is typically done by cell projection or ray casting. A number of papers [13, 8, 6, 7, 4, 2, 1, 5] have presented results of parallel volume rendering of unstructured data, but their work has not utilized hardware graphics. A parallel algorithm for unstructured data using PixelFlow, a sort-last graphics architecture, is described in [18]. A multi-threaded algorithm for tetrahedral unstructured grids on a two-CPU system was reported on in [19]. In this paper, we describe the results of parallelizing the high-accuracy (HIAC) volume rendering system described in [16] for unstructured data (characterized by meshes of either tetrahedra or mixed cell types) using hardware rendering on an SGI Onyx2[3].

2 OVERVIEW OF HIAC

HIAC operates in two different modes, *tetrahedra* or *zoo-mesh*, corresponding to the type of the input data. Zoo-mesh element may be quadrilateral-faced hexahedra or "bricks," pentahedra (either triangular prisms or quadrilateral-based pyramids), or tetrahedra. Polyhedron faces may be planar or non-planar, possibly resulting in nonconvex cells. We parallelized the code for both the zoo mesh and tetrahedron only modes, and compared their performance on the twisted curvilinear grid shown in figure 11.

[1]University of California, Davis

[2]Lawrence Livermore National Laboratory

[3]The Onyx2 is a multiprocessor machine with a shared-memory architecture. The machine we used for testing has 48 250 MHZ IP27 MIPS R10000 Processors, a main memory size of 15872 Mbytes, a 32 Kbyte instruction cache and a 32 Kbytes data cache size per processor.

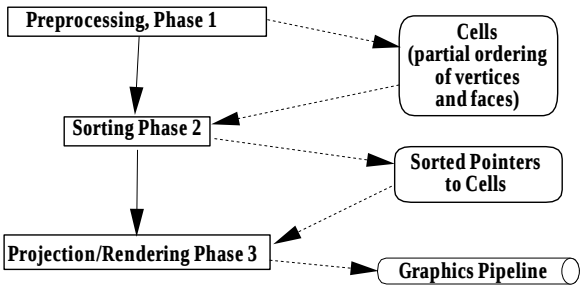

Figure 1: Basic HIAC operation in serial. Solid arrows show control flow; dashed arrows are for data evolution.

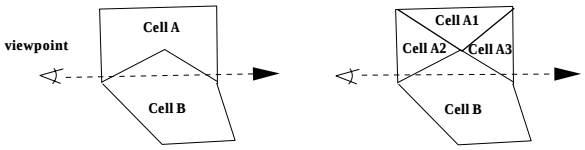

Figure 2: An example of how a twisted face between cells A and B cause their front/back relationship to be indeterminate. Note that there would be no problem with the twisted face if the viewpoint were looking up in the picture from the bottom of cell B. One way to fix the problem is by subdividing cell A as shown.

When operating in serial, as shown in Figure 1, HIAC execution is characterized by three phases, which are the *preprocessing* phase, the *visibility sort* phase, and the *projection and rendering* phase. In the preprocessing phase, the entire data set is read from disk into core memory. In core memory, cells are stored in an internal representation that maintains information such as pointers to vertices and faces as well as adjacency information. Prior to actually sorting the cells, a partial ordering on the cells is generated by marking each cell face with an arrow. The arrow indicates which of the two cells sharing the face is in front with respect to the viewer position. Cells with non-planar faces may cause problems since a ray from the viewpoint may enter and exit a cell, pass through another cell and then reenter the first cell (see Figure 2) causing a visibility ordering cycle. Such cells are tested to see if they do cause this problem and, if so, they are subdivided into tetrahedra to eliminate visibility-ordering cycles (see [9]).

The cells are visibility sorted from back to front using the partial ordering. The sorting is done by one of two methods depending on user preference for speed or accuracy. It may be done using *Meshed Polyhedron Visibility Ordering–Non-Convex* (MPVONC) algorithm [14], which is fast but does not guarantee an accurate

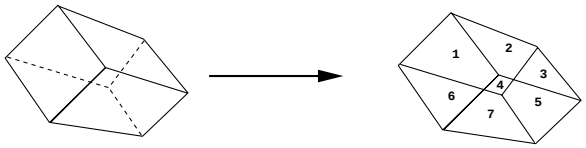

Figure 3: HIAC's projection and slicing of a cell by its edges into polygons in the view plane.

sort. Another sorting algorithm that may be used is the *Scanning Exact MPVO* (SXMPVO) [3] algorithm, a modification of the MPVONC algorithm that is exact but slower.

In the projection and rendering phase, the sorted cells are projected onto the screen in back to front order using the projection scheme described in [16] and [9]. As they are projected, OpenGL may be used to draw and composite the resulting polygons, or the rendering may be done in software. The software rendering algorithm is described in Section 4.

When OpenGL is used, HIAC first flattens each cell into a number of view plane polygons, not necessarily triangular, in a manner similar to that for tetrahedra in the Shirley-Tuchman method [12] (see Figure 3). For cells other than tetrahedra, projections topologically equivalent to the one in Figure 3 or to one of the two other non-degenerate perspective projections of a cube are called "simple" cases as such cases generate predictable lists of triangle fans as in [12] and [11]. In other cases, the polygons created are a partitioning of the projection of the cell by the projection of each of its edges on the view plane. This partitioning is done by adding the projected edges one by one to a winged-edge structure (described in [10]) for the view plane. In this manner, HIAC can handle polyhedra ranging from tetrahedral cells to more general zoo-mesh cells.

The computational geometry for the general cell projection algorithm is complicated and slow, so the three simple hexahedron projection cases take much less time. Hexahedral cells with nonplanar faces have some viewpoints where their projections are simple, and others where they are not. The viewing angle also affects whether a cell causes a visibility cycle and must be subdivided into tetrahedra. Therefore, the performance of the zoo element mode varies substantially with viewpoint changes, while the tetrahedron only mode is less sensitive to such changes.

To render each polygon thus created, HIAC calculates the exact color and opacity of the ray segment through each vertex of each polygon, based on the opacity transfer function of the cell and the length of the ray segment. If the transfer functions that map scalar field values to opacity and color densities are linear and the scalar field being imaged also varies linearly along the ray segment, then the integrated opacity is a quadratic along the ray segment and the color integral can be computed analytically (see [15] for the solution). OpenGL calls can be then used to render the polygon, taking advantage of graphics hardware for polygon rasterization and interpolation of color between polygon vertices, as well as back-to-front fragment composition.

If the opacity varies linearly along the ray segments, 2D texture maps may be used as in [16] together with the compositing functions present in most graphics cards to compute the opacity contribution of the ray segment to fairly high accuracy. The calculation is done by preloading the texture map with the function

$$\alpha(u, v) = 1 - e^{-uv} \tag{1}$$

Equation 1 is the integral for the opacity of a line segment from z_1 to z_2, of length $u = z_1 - z_2$ and with linearly varying opacity density between τ_{z_1} and τ_{z_2}, with $v = \tau_{avg} = \frac{1}{2}(\tau_{z_2} + \tau_{z_1})$. Graphics compositing hardware then blends the interpolated polygon color $Color_{New}$ with the frame buffer color $Color_{Old}$ using the standard equation

$$Color_{Out} = (1 - \alpha)Color_{Old} + \alpha Color_{New} \tag{2}$$

HIAC uses linear or bilinear interpolation of vertex colors across polygons, and equations 1 and 2 to correctly composite polygons from back to front in hardware. The values of u and v are computed at the polygon vertices and the texture mapping hardware interpolates u and v across the polygons. Since each of the view-plane polygons, as shown in Figure 3, is in the projection of a single front-facing face of the cell and a single back-facing face, the ray segment length $z_2 - z_1$ varies linearly across the polygon. For a linear variation of τ within the cell, τ_{avg} also varies linearly across the polygon. Therefore, the texture coordinate interpolation gives the correct opacity α for each pixel. However, the integrated ray color varies in a more complicated way across the polygon, so the hardware integrated color is only approximately correct.

Further compromises of image quality for speed may be made when operating using hardware projection. For example, the user may choose to not calculate the color integral but instead, for example, use the average of the colors at the near and far ends of the ray segment. One may eliminate the texture mapping, and instead have the hardware interpolate the vertex opacities across the polygons. Such approximations can lead to artifacts in the image, but are included in HIAC to allow flexibility to the user. For completely accurate colors, one must do the analytic integral for each pixel's ray segment in software. The parallelization of the software rendering algorithm is described in section 4. More details about the serial version of HIAC are given in [16] and [9], while [17] gives a detailed discussion of the current state of HIAC and of the SXMPVO sorting algorithm.

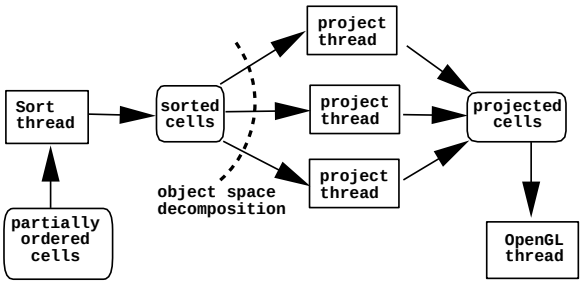

Figure 4: HIAC operation in parallel mode.

3 PARALLEL ALGORITHMS FOR HARDWARE RENDERING

3.1 Parallel Cell Projection

When HIAC is run in serial with hardware rendering, the projection step of the algorithm takes most of the time. Therefore, we focused our efforts on parallelizing it. The output of the sorting phase (the input to the projection phase) is an array of ordered cells. This array can be decomposed into subarrays and fed to the projection phase as independent work quanta as long as the overall

```
typedef struct {
  int rk_class;
  float texture[MAXPTS*2];
  float   color[MAXPTS*4];
  float  vertex[MAXPTS*3];
} splat;
```

Figure 5: Data structure to support tetrahedra mode thread communication.

ordering is maintained, which is achieved by thread communication discussed in Section 3.2.

During HIAC parallel operation, each cell moves through a pipeline as in Figure 4. The sorting thread proceeds to place the cells' indices into an array in back-to-front order as they are discovered by the visibility ordering algorithm. When the sorting thread finishes "enough" cells (this is a tuning parameter) to constitute a work quantum, the projection threads begin to project the sorted cells. The projection threads place their resulting vertex, color, and texture information into another array. The render thread pulls this information and issues OpenGL calls.

3.2 Communication

Pthreads were chosen over MPI because the Pthreads library uses shared-memory processing, which is natural to the IRIX operating system on which HIAC is currently being developed. Furthermore, process-level parallelism using MPI would dramatically increase message overhead due to its use of network protocols for messaging. MPI would, however, allow for easier porting from SGI to large-scale distributed-memory architecture cluster configurations (such as Lawrence Livermore National Laboratory's ASCI White machine).

When operating in tetrahedra mode, two sets of global buffers, whose access is restricted by semaphores, are used for communication between phases. The first of these is composed of two integer arrays, called the sort_projection buffers, which are used for transferring cell indices from the sort thread to the projection threads. One of the projection threads, the Communicator, is selected to communicate with the sort and render threads. The sorting thread sorts a global array of cells in visibility order and stores the sorted indices into one of these buffers. Once this array is full, the sort_projection buffers are swapped and the Communicator wakes up the projection threads, which begin projecting these cells. The projection threads each have a non-overlapping section of the sort_projection buffer that they pull their work from in order to gain locality of memory references for each thread. The threads project the tetrahedra onto the view plane and decompose their projections into triangles. For each tetrahedron they also record the Shirley-Tuchman case [12] as well as all of the vertex, color, and texture information needed to render the tetrahedron. This information is stored in the appropriate data structure (see Figure 5) in one of two projection_render arrays. While the projection threads are filling one projection_render array, the render thread is using the information stored in the other array to make the appropriate OpenGL calls.

During zoo-mode operation, communication between sort and projection threads is handled by the sort thread. When the sort thread finishes sorting "enough" cells, it obtains a workQuantum (see Figure 6) from the global pool and places a reference to this quantum into two global arrays, one of which is used to characterized the work to be done by the projection threads one of which

```
typedef struct {
  char doProjectFlag;
  long startCell, endCell, pcbIndex;
  long ttStart, ttEnd, pcbIndex;
  char doneProjectingFlag,
     inUseFlag, isLastFlag;
  long vvIndex, vvMax,
     polyIndex, polyArraySize;
  GLfloat *mTexArray, *mVertexArray,
     *mColorArray;
  int *polyVertexCounts;
} workQuantum;
```

Figure 6: Data structure to support zoo-mesh mode thread communication.

is used to characterize the work of the rendering thread. The sort thread next fills in the start and end cell information for the work quantum and then awakens a single projection thread to process it. The projection thread performs the projection calculations on the cells referenced by the work quantum and stores the information from its calculations on each cell in arrays within each work quantum. When the work quantum is complete, the projection thread marks the doneProjectingFlag for the work quantum and goes to sleep waiting for another work quantum. The render thread continually polls the doneProjectingFlag of the next work quantum in its work pile and, when this flag is set, the render thread makes the OpenGL calls required to display the final image.

4 PARALLEL ALGORITHMS FOR SOFTWARE RENDERING

HIAC creates its most accurate images by performing a separate exact analytic or numerical integration for each of the three primary colors for each pixel that a cell projects onto. This can only be done in software. The results of the ray integrations are stored in 3 separate frame buffers, one each for the red, green and blue channels. Therefore, if cell i projects onto p_i pixels, for n cells a total of $\Sigma_{i=1}^{n}(3p_i)$ integrations are needed. For large data sets, this may take on the order of hours. Therefore, we investigated a parallel algorithm on an IBM SP2 using MPI.

In this algorithm, the entire data set is first distributed to all the nodes of the SP2, then for each view point, the nodes run a distributed k-d partitioning algorithm to create load balanced partitions of the data at each node. No data actually moves; only bounding box statistics are exchanged until a satisfactory balance is achieved. Then each node calculates a visibility ordering and then performs volume rendering integrations on its partition of the data, resulting in a tile of the image in the three frame buffers residing on each node. Threading is not used. The green frame buffer tiles are sent to a "green master" node, red to a "red master," and blue to the "blue master" node, via the interconnect switch. The green master node then assembles the final green image, etc. Finally, the green, blue and red masters send their images to the overall master node, which then accumulates the three frame buffers into the final RGB image and writes it to disk.

number of nodes	load balance time	max sort render time	gather time	overall time
4	0.09	39.17	0.43	39.69
8	0.09	13.30	2.40	15.79
16	0.08	6.09	2.75	9.64
32	0.08	2.64	1.52	4.24
64	0.08	3.3	3.09	6.47
128	0.08	1.02	1.05	2.15

Table 1: Timings from SP2 environment. Data set size: 600,000 tetrahedra, all times in minutes.

		Seconds
Number of Projection Threads	1	81.6
	2	41.2
	3	28.1
	4	21.6
	6	15.4
	8	12.1
	10	10.2
	12	9.3
	18	7.6
	22	7.7

Table 2: Overall volume rendering times using zoo mode parallelism for a data set with 299,000 hexahedral cells, which after view-dependent subdivision had 308,190 cells. Times are shown for the use of various numbers of projection threads (using method M0 and *casification*). In addition to the projection threads, one thread is used for sorting, MPVONC in this case, and one for making the OpenGL calls. All timings are in seconds.

5 RESULTS

5.1 Results on the SP2

The results of the SP2 software experiment are shown in Table 1. The gathering operation is the bottleneck and shows up at around 32 processors. A more sophisticated gathering algorithm should postpone the flattening of the speed-up curve to 64-128 processors. Load balancing time is minimal and requires little communication overhead. A good speed up is obtained by partitioning the sorting and rendering – the superlinear speedup is accounted for by the tiling of the original NNS sort, as described in [16]. Overall, the results were satisfactory and should scale to much larger data sets.

5.2 Parallelism Speedup on the SGI

To determine the speedup from parallelizing HIAC, we measured the execution time of HIAC from the moment the sort started to the time the last cell was projected using various numbers of projection threads. We used the MPVONC sorting algorithm for parallel measurements.

5.2.1 Zoo-Mesh Mode

In order to compare the performance of the tetrahedron only and zoo element modes, we created artificial data sets of varying sizes, by wrapping a curvilinear grid around a cylinder, as shown in Figure 11. The twisting of the spiral created four non planar faces in every hexahedron, exercising the view-dependent subdivision in the zoo element mode.

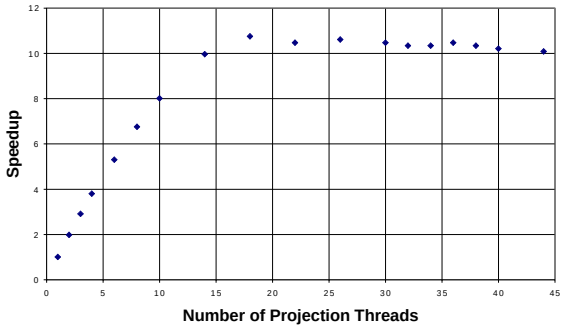

Figure 7: Program speedup using zoo-mesh cell projection on a dataset of 308,190 cells.

We expected the zoo element mode to generate less triangles and put less burden on the graphics pipeline. For example, the hexahedron projection shown in figure 3 can be drawn using two triangle fans, with a total of 12 non-overlapping triangles. On the other hand, when subdivided into six tetrahedra and rendered with the Shirley-Tuchman method, it requires six triangle fans, with a total of 22 triangles. In some places as many as four of these triangles overlap, adding to the OpenGL fragment count, as well as to the vertex and triangle count.

Our tests showed that the zoo element mode required approximately 2/3 as many triangles as the tetrahedron only mode. This is not as good as the 12/22 ratio above because the non-planar faces of some cells caused visibility cycles, forcing them to be subdivided anyway. The best performance for the zoo element mode was faster than the tetrahedron only mode, by approximately the ratio of triangles rendered, but this was achieved at the expense of using many more processors, because the general cell projection method was much slower. The relation of rendering time to triangle counts may not be meaningful, however, because neither projection method was able to saturate our graphics hardware pipeline, no matter how many projection processors we used.

When operating in zoo-mesh mode, Figure 7 shows that for up to ten threads, speedup (as measured by execution time using one thread divided by time using n threads) is very near linear. (Linear speedup is the optimal result for the producer–consumer type of parallelism used in zoo-mesh mode). Adding more threads results in diminishing returns, until at 18 threads the curve flattens out and there is actually a small slowdown. We believe this decline is due to communication costs over the SGI NUMA (see Section 5.2.3).

The optimum number of cells to choose per work quantum is a tuning parameter in the zoo code. It is best to choose neither a very small nor large quantum size. Small quanta are finished very often; therefore, the sorting and projection threads spend a significant portion of their time contending for mutexes in order to establish the workload distribution. For very large quanta, which approach in size the order of the total workload, the projection threads waste time waiting for the sorting thread to finish each work quantum so that they can begin their work.

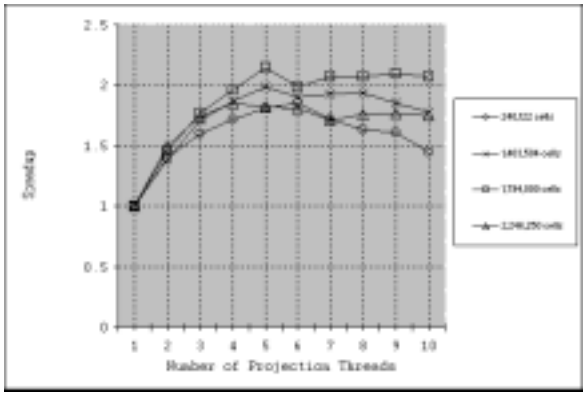

Figure 8: Improvement in program performance with increasing parallelism on tetrahedral data sets of different numbers of cells.

number of project threads	240,122 cells "f117"	1,403,504 cells "fighter"	1,794,000 cells	2,246,250 cells "helix"
1	3.49	27.66	27.26	32.68
2	2.49	19.79	18.64	21.66
3	2.14	16.14	15.40	18.78
4	2.03	14.80	13.88	17.70
5	1.93	13.93	12.69	17.97
6	1.89	14.59	13.67	18.02
7	2.03	14.36	13.13	19.15
8	2.14	14.26	13.32	18.58
9	2.17	14.94	13.00	18.45
10	2.41	15.54	13.25	18.66

Table 3: Timings for Tetrahedra Mode. All timings are in seconds.

5.2.2 Tetrahedra Mode

Figure 8 and Table 3 show that increasing the number of projection threads to five or six results in the optimum tetrahedra mode runtime, which is roughtly half the run time of running the code with only one projection thread. (The "fighter", "f117", and "helix" images are shown in Figures 10, 9 and 11 respectively.) We believe that, when running in parallel tetrahedra mode, the limitation on HIAC performance is the memory bandwidth of the NUMA to send color and vertex information from the projection threads to the rendering thread. Since the zoo code requires more computational effort, this effect is not seen until more threads are used and more memory bandwidth is generated by the projection threads. This explains why the speedup curve for the tetrahedral code flattens out for fewer processors than for the zoo mesh code.

5.2.3 Peculiarities of the SGI Memory Architecture

SGI multiprocessor machines use a non-uniform memory architecture (NUMA), which means that the time it takes a process to access memory is proportional to the physical distance between the processor and memory. During parallel execution of the HIAC program, data needed by a processor are not necessarily physically close to the processor in memory, but may be several "hops" away. The number of hops a memory reference needs to travel to fulfill a CPU request will tend to increase as the number of processors working on the data set increases. There is an inherent limit to speedup on the NUMA due to these extra communications costs. Hofsetz and Ma [5] implemented a software parallel cell profection algorithm with pthreads on an SGI Origin 2000, which has the same processor and interconnect architecture as our Onyx2. Their speed-up curve exhibited flattening at about the same number of processors as does Figure 7.

Acknowledgement

We are grateful for the expert technical advice and contributions of Randy Frank of Lawrence Livermore National Laboratories. The "fighter" data set was graciously shared with us by David Marcum and the Computational Simulation and Design Center at the Mississippi State University Engineering Research Center. The "f177" data set came to us through the kindness of Robert Haimes of MIT.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[1] Judy Challinger. Parallel volume rendering on a shared-memory multiprocessor. 1991.

[2] Judy Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 81–88, November 1993.

[3] Richard Cook, Nelson Max, Claudio Silva, and Peter Williams. Efficiently sorting zoo-mesh data sets. Technical Report UCRL-ID-143126, Lawrence Livermore National Laboratories, Livermore, CA, June 2001.

[4] C. Giertsen and J. Petersen. Parallel volume rendering on a network of workstations,. *IEEE Comp. Gr. and Applic.*, 13(6):16–23, 1993.

[5] C. Hofsetz and K.-L. Ma. Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000. In *Third Eurographics Workshop on Parallel Graphics and Visualization*, 2000.

[6] K-L Ma and T. W. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimesnional unstructured data. *IEEE 1997 Symp. on Parallel Rendering*, pages 95–104, Oct. 1997.

[7] K-L Ma and T. W. Crockett. Parallel visualization of large-scale aerodynamic calculations: A case study on the cray t3e. *1999 IEEE Parallel Visualization and Graphics Symp.*, pages 15–20, Oct. 1999.

[8] Kwan-Liu Ma. Parallel volume ray-casting for unstructured grid data on distributed memory architectures. *IEEE 1995 Parallel Rendering Symposium*, pages 23–30, October 1995.

[9] Nelson Max, Peter Williams, and Claudio Silva. Approximate volume rendering for curvilinear and unstructured grids by hardware-assisted polyhedron projection. *International Journal of Imaging Systems*, 11:53–61, 2000.

[10] Joseph O'Rourke. Computational geometry in C. 1993. ISBN 0-521-44034-3.

[11] Greg Schussman and Nelson Max. Perspective volume rendering using triangle faces. In *Proceedings of the International Workshop on Volume Graphics*, Stony Brook, NY, June 2001.

[12] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:63–70, November 1990.

[13] P. Williams. Parallel volume rendering finite element data. In Thalmann and Thalmann, editors, *Communicating with Virtual Worlds*, pages 473–484. Springer Verlag, 1993.

[14] Peter Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[15] Peter Williams and Nelson Max. A volume density optical model. *ACM Workshop on Volume Visualization*, pages 61–68, October 1992.

[16] Peter Williams, Nelson Max, and Clifford Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January-March 1998.

[17] Peter L. Williams and Nelson L. Max. The LLNL high accuracy volume renderer for unstructured data: Capabilities, current limits, and potential for ASCI/VIEWS deployment. Technical Report UCRL-ID-144107, Lawrence Livermore National Laboratories, Livermore, CA, June 2001.

[18] C. M. Wittenbrink. Irregular grid volume rendering with composition networks. In *Proceedings of IS&T/SPIE Visual Data Exploration and Analysis*, volume 3298, pages 250–260. SPIE, January 1998. Available as Hewlett-Packard Laboratories Technical Report HPL-97-51-R1.

[19] Craig M. Wittenbrink, M.E. Goss, and H. Wolters. Interactive unstructured volume rendering and classification. Technical Report HPL-2000-13, Hewlett-Packard Laboratories, Palo Alto, CA, January 2000. Submitted to the Proceedings of Dagstuhl, Germany workshop on scientific visualization, May 2000.

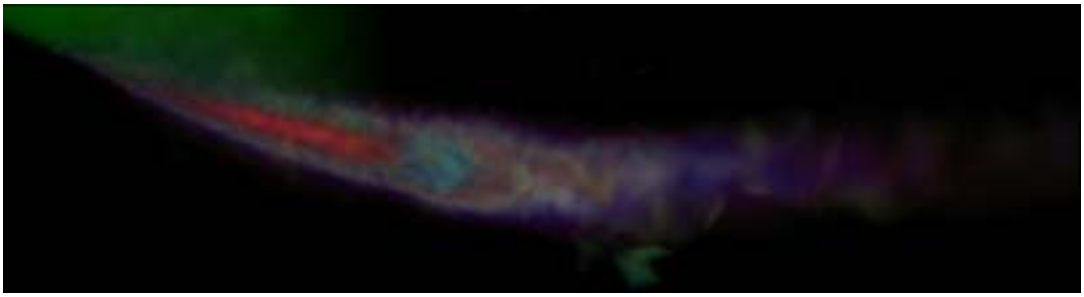

Figure 9: The “f117” dataset image.

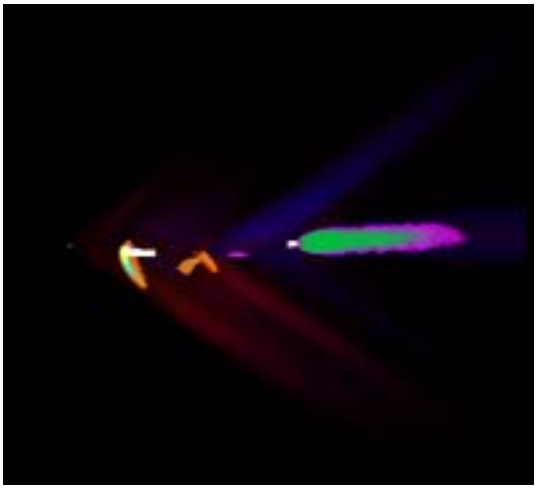

Figure 10: The “fighter” data set image.

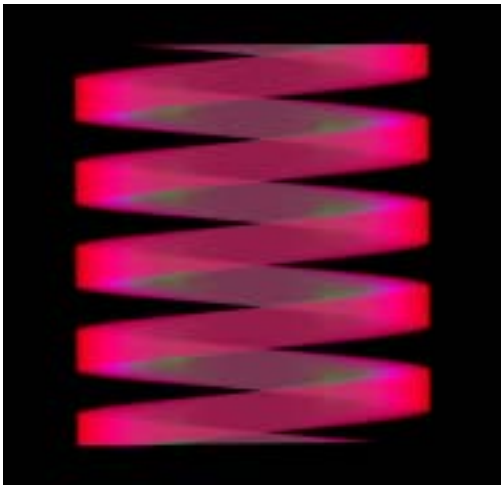

Figure 11: The “helix” dataset image.